# Tutorials for queueing simulations

N.D. Van Foreest and E.R. Van Beesten

January 10, 2020

CONTENTS

# 1 TUTORIAL 1: EXPONENTIAL DISTRIBUTION

The aim of this tutorial is to empirically show a fascinating fact: even for very small populations in which individuals decide independently to visit a server (a shop, a hospital, etc), the exponential distribution is a good model for the inter-arrival times as seen by the server. We will develop a simulation to motivate this 'fact of nature'. In particular, our aim is to build the analogues of Figs. 1 to 3 in terms of cdfs instead of pdfs.

I expect that you will be able to cover up to about Section 1.5 during the tutorial. The rest of the sections are homework. All solutions are at the end of this document. I also make the code available on github in the form of one complete Python program and as a Jupyter notebook.

If you want to increase your probabilistic intuition, coding skills, and knowledge of data analysis, try to do all exercises by yourself. This is quite challenging, if you get stuck, check the solutions. Otherwise, if this is too much of challenge, just start from the solutions or the Jupyter notebook.

## 1.1 *Background*

We discuss an example to intuitively see how the exponential distribution originates. Consider a group of $N$ patients that have to visit a hospital for regular checkup, for instance, after each visit they have to see an MD somewhere between 4 to 6 weeks later. Let us assume that the inter-arrival times $\{X_k^i, k = 1,2,\ldots\}$ of patient $i$ are reasonable well modeled as uniform distributed between 4 and 6 weeks, i.e., $X_k^i \sim U[4,6]$. Then, with $A_0^i = 0$ for all $i$, define

$$A_k^i = A_{k-1}^i + X_k^i = \sum_{j=1}^{k} X_j^i \tag{1.1}$$

as the arrival moment of the $k$th visit of patient $i$.

Now the hospital doctor 'sees' the superposition of the arrivals of all patients. One way to compute the arrival moments of all patients together is to put all the arrival times $\{A_k^i, k = 1,\ldots,n, i = 1,\ldots,N\}$ into one set, and sort these numbers in increasing order. This results in the (sorted) set of all arrival times $\{A_k, k = 1,2,\ldots\}$ as seen by the doctor. Taking $A_0 = 0$, it follows hat

$$X_k = A_k - A_{k-1}, \tag{1.2}$$

is the inter-arrival time between the $k-1$th and $k$th patient as seen by the doctor. Thus, with this procedure, starting from inter-arrival times of individual patients, we can construct inter-arrival times of the entire population as seen by the doctor.

To plot the empirical distribution function, of $\{X_k\}$, we just count the number of inter-arrival times smaller than time $t$ for any $t$. In other words, we compute the empirical distribution of $\{X_k\}$ as

$$\mathsf{P}_n(X \le t) = \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}_{X_k \le t},$$

where the *indicator function* is $\mathbb{1}_{X_k \le t} = 1$ if $X_k \le t$ and $\mathbb{1}_{X_k \le t} = 0$ if $X_k > t$.

In the three panels in Figure 1 we show the emperical distribution for three different cases. In the first example, we take $N = 1$ patient and let the computer generate $n = 100$ uniformly distributed numbers on the set $[4,6]$. Thus, the time between two visits of this single patient is somewhere between 4 and 6 hours, and the average inter-arrival times $\mathsf{E}[X] = 5$. For the second simulation we take $n = 100$ visits for $N = 3$ patients, and in the third, $N = 10$ patients. We add as a reference (the continuous curve) the density of the exponential distribution $\lambda e^{-\lambda t}$ for $\lambda = 1/5$, $\lambda = 3/5$ and $\lambda = 10/5$, respectively. (Recall that when one person visits the doctor with an average inter-arrival time of 5 hours, the arrival rate is 1/5. Hence, when $N$ patients visit the doctor, each with an average inter-arrival time of 5 hours, the total arrival rate as seen by the doctor must be $N/5$.)

In Figure 2 we extend the number of visits to $n = 1000$, and in Figure 3 we take the inter-arrival times to be normally distributed times with mean 5 and $\sigma = 1$.

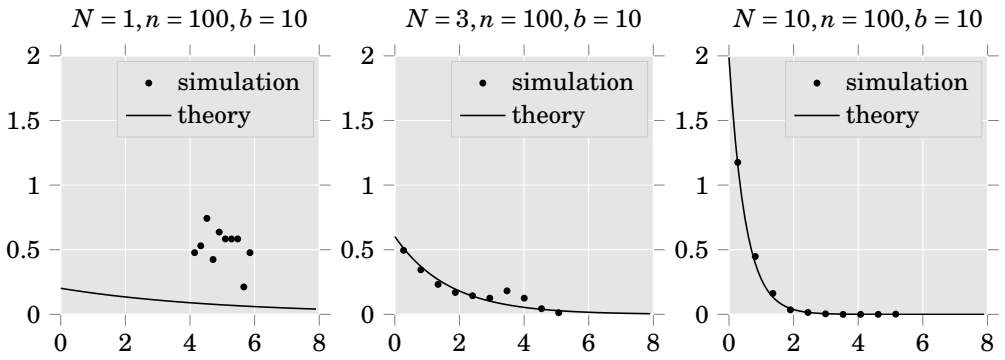


Figure 1: The inter-arrival process as seen by the doctor owner. Observe that the density $\lambda e^{-\lambda x}$ intersects the $y$-axis at level $N/5$, which is equal to the arrival rate when $N$ persons visit the doctor. The parameter $n = 100$ is the simulation length, i.e., the number of visits per patient, and $b = 10$ is the number of bins to collect the data, see Remark 1.1.

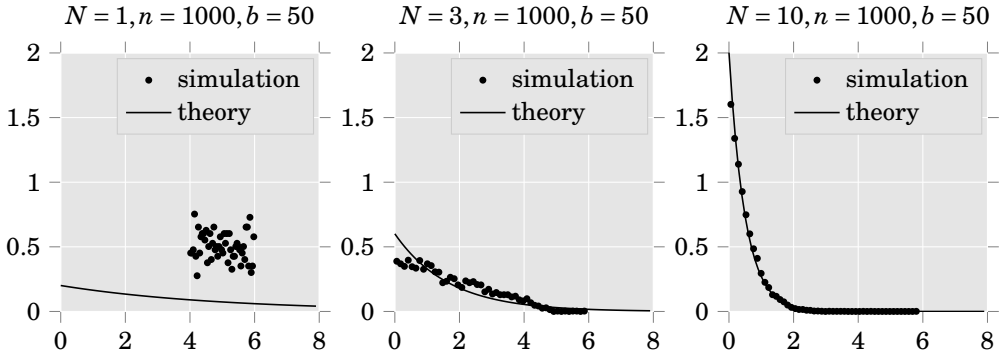


Figure 2: Each of the $N$ patients visits the doctor at uniformly distributed inter-arrival times, but now the number of visits is $n = 1000$.

As the graphs show, even when the patient population consists of 10 members, each visiting the doctor with an inter-arrival time that is quite 'far' from exponential, the distribution of the inter-arrival times as observed by the doctor is very well approximated by an exponential distribution. Thus, it appears reasonable to use the exponential distribution to model inter-arrival times of patients for systems (such as a doctor, or a hospital or a call center) that handle many (thousands of) patients each of which deciding independently to visit the system.

**Remark 1.1.** The computations of the bins in Figs. 1 to 3 is a bit subtle, in particular the height. The procedure works as follows. Let $a = (X_1, \ldots, X_n)$ denote the simulated inter-arrival times. Define the width of a bin as $\delta = (\max\{a\} - \min\{a\})/b$, where $b$ is specified by the user, for instance $b = 10$. Thus, if $b = 10$, the total range of observations is divided in $b = 10$ cells. Then the height of the $i$th bin is computed as $h_i = n_i/(n\delta)$ where $n_i$ is the number of observations in the $i$th cell. Observe that then $\sum_{i=1}^{b} h_i \delta = 1$.

## 1.2 *Simulations*

**1.1.** Make a plan of the steps you have to carry out to make an analogue of Figure 1.1 of the queueing book in terms of a cdf. In the next set of exercises we'll carry out these steps. So please do not read on before having thought about this problem, but spend some 5 minutes to think about how to approach the problem and how to chop it up into simple steps. Then organize the steps into a logical sequence. Don't worry at first about how to convert your ideas into computer code. Coding

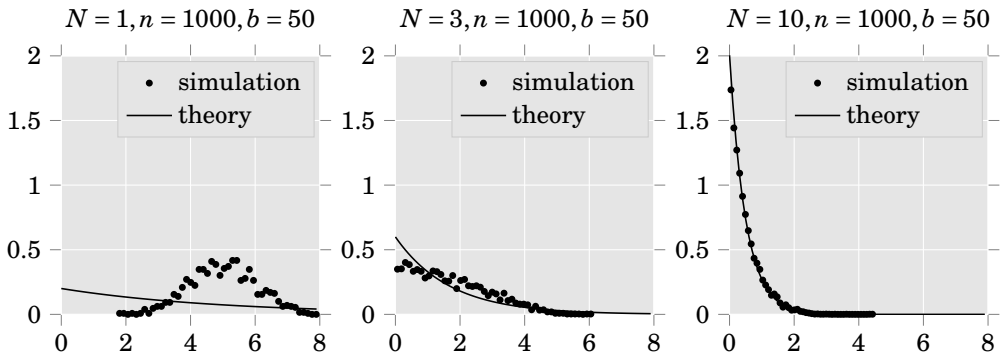


Figure 3: Each of the $N$ patients visits the doctor with normally distributed inter-arrival times with $\mu = 5$ and $\sigma = 1$.

is a separate activity. (As a matter of fact, I always start with making a small plan on how to turn an idea into code, and I call this step 'modeling'. Typically this is a creative step, and not easy.)

We need some python libraries to make our life a bit easier. You should copy this code into your editor.

```
import numpy as np
import scipy
import matplotlib.pyplot as plt

plt.ion()  # you can skip this, I only use it for testing purposes so that the computer skips mak

# this is to print not too many digits to the screen
np.set_printoptions(precision=3)

# fix the seed
scipy.random.seed(3)
```

You need to know what the seed is of random number generator; search for it on the web.

### 1.3 *Empirical distributions*

An important step in a simulation is to analyze the output data. For this we need the *empirical distribution*, which is defined for a set of numbers $\{a_1, \ldots, a_n\}$ as

$$F(x) = \frac{\#\{i : a_i \leq x\}}{n}. \tag{1.3}$$

We note that the computation of $F$ is much more interesting (and challenging) than you might think[1].

Before designing an algorithm to compute the emperical distribution, it is best to first make an emperical distribution by hand, and then formalize our steps.

**1.2.** Suppose you are given the following sample from a population: $3.2, 4, 4, 1.3, 8.5, 9$. Make the emperical distribution. Then analyze all steps you took to make the empirical distribution function.

Attempt to turn your ideas into an algorithm. You'll see that this is harder than you might have guessed before you really tried. After some thought and trying, read (and study) the code in the solution.

[1] If you search the web, you will see that computing the empirical *density* function, rather than the *distribution* fundction, is even more challenging.

**1.3.** The method provided by the (solution of the) previous exercise is simple, but not completely correct. What is wrong?

**1.4.** Take $s$ as the sorted version of the list $a$. Make a plot (by hand) of $s$, that is, plot the points $(i, s_i)$. Observe the crucially important fact that the function $i \to s_i$ is the inverse of the (unnormalized) distribution $F$.

**1.5.** Find now a way to invert $i \to s_i$, normalize the function to get a distribution, and make a new plot.

**1.6.** In the previous exercise (read the solution), we start $y$ with 1 and end with `len(a)+1`. Why is that?

You should know that `for` loops in python are quite slow (and `for` loops in R seem to be really dramatic). For large amounts of data it is better to use `numpy`.

**1.7.** Use the `numpy` functions `np.arange` and `np.sort` to speed up the algorithm of the previous exercise.

With the algorithm of Exercise 1.7 we can compute and plot a distribution function of inter-arrival times specified by a list (vector, array) $a$. For our present goals this suffices.

If you like details, you should notice that our plot of the distribution function is still not entirely OK: the graph should make jumps, but it doesn't. Moreover, our cdf is not a real function, it can be of the form $x = (1,1,3)$, $y = (0, 0.5, 1)$. In the rest of this subsection we repair these points. You can skip this if you are not interested.

**1.8.** Read about the `drawstyle` option of the `plot` function of `matplotlib` to see how to make jumps in plots.

**1.9.** But now we still have vertical lines. To remove those, check how to use `hlines`.

**1.10.** Finally, we can make the computation of the cdf significantly faster with using the following `numpy` functions.

1. `numpy.unique`
2. `numpy.sort`
3. `numpy.cumsum`
4. `numpy.sum`

How can you use these to compute the cdf?

## 1.4 *Simulating the arrival process of a single patient*

With the above work we have the tools to compute the distribution function of a number of measurements or observations of some (stochastic) process. In queueing theory we are mostly interested in inter-arrival times and service times of customers. In this section we focus on the inter-arrival times of recurrent visits of a single patient at a doctor, in weeks say.

We first use `scipy` to generate sequences of i.i.d. random numbers with a given uniform distribution. Then we compute the emperical cdf for these data and compare it graphically the cdf of the uniform and the exponential distribution. A more formal method to compare cdfs is by means of the Kolmogorov-Smirnov statistic (see Wikipedia); we develop this concept in passing.

**1.11.** Read the documentation of the `uniform` class of the `scipy.stats` module[2]. Check in particular the documentation of the `rvs()` function. Use the documentation to see how to Generate 3 random numbers uniformly distributed on $[4,6]$. Print these numbers to check whether you get something decent.

[2] When coding you should develop the habit to look up things on the web

**1.12.** Generate $L = 300$ random numbers $\sim U[4,6]$ and make a histogram of these numbers. For this, you can use the `hist` function of `matplotlib`; see the web for the documentation, in particular the `density` and `cumulative` options of the `hist` function are useful for our purposes. Then compute the empirical distribution function of these inter-arrival times, and plot the cdf.

**1.13.** We would like to numerically compare the empirical distribution of the inter-arrival times to the theoretical distribution, which is uniform in this case. For this we can use the Kolmogorov-Smirnov statistic. Try to come up with a method to compute this statistic, and then compute it. Look up a table with critical values for the Kolmogorov-Smirnov test statistic. Given a 5% confidence level, do we reject the hypothesis that our sample is drawn from a $U[4,6]$ distribution?

You might find the following functions helpful (read the documentation on the web to see what they do):

1. `numpy.max`
2. `numpy.abs`
3. `scipy.stats.uniform`, the `cdf` function.

**1.14.** Finally, plot the empirical distribution and the exponential distribution with mean $\lambda = 1/5$ in one graph. (Why do we take $\lambda = 1/5$?) The relevant `scipy` functions can be found with searching on `scipy.stats.expon`.

Explain why these graphs are different. In passing, compute the KS statistics to compare the simulated inter-arrival times with an exponential distribution. Do we reject the hypothesis that our sample is drawn from this exponential distribution?

## 1.5 *Simulating many patients*

We would now like to simulate the inter-arrival process as seen by a doctor that serves many recurring patients. For ease, we call this the *merged*, or *superposed*, inter-arrival process.

Making the merged process out of a number of individual patient arrival process turns out to an interesting algorithmic challenge. Thus, we start with a numerical example with two patients, and organize all steps we make to compute the empirical distribution of the merged inter-arrival process. Then we make an algorithm, and scale up to arbitrary numbers of patients.

**1.15.** Suppose we have two patients with inter-arrival times $a = [4,3,1.2,5]$ and $b = [2,0.5,9]$. Make by hand the empirical cdf of the merged process. Then summarize the steps you took in the process.

**1.16.** The steps of the previous exercise can be summarized by the following code:

```
from itertools import chain

def superposition(a):
    A = np.cumsum(a, axis=1)
    A = list(sorted(chain.from_iterable(A)))
    return np.diff(A)
```

Note that the input `a` in the function `superposition` is a matrix of inter-arrival times of several patients.

Study this code by reading the documentation (on the web) of the following functions, `numpy.cumsum`, in particular read about the meaning of axis, `itertools.chain.from_iterable`, and `numpy.diff`.

**1.17.** Generate 100 random inter-arrival times $\sim U[4,6]$ for 3 individual patients. Then use the code above to compute the inter-arrival times of the merged stream. Plot the emperical cdf of the merged inter-arrival times, and compare the emperical cdf to the exponential distribution with the correct mean (which should by $\lambda = 3/5$). Also compute the Kolmogorov-Smirnov statistic for this case.

What is the effect of increasing the number of patients from $N = 1$ to $N = 3$?

**1.18.** Compare the empirical distribution of the inter-arrival times generated by $N = 10$ patients to the exponential distribution (What is now the appropriate arrival rate:). Make a plot, and explain what you see.

**1.19.** Do the same for $N = 10$ patients with normally distributed inter-arrival times with $\mu = 5$, and $\sigma = 1$. For this use `scipy.stats.norm`. What do you see? What is the influence of the distribution of inter-arrival times of an individual patient?

**1.20.** If $\mu = \sigma = 5$ then the analysis should break down. What happens if you try this settting? If you don't get real strange results, the code itself must be wrong[3].

## 1.6 *Memoryless property*

It is well known that the exponential distribution has the memoryless property, that is, when $X$ has an exponential distribution,

$$\mathsf{P}(X > s+t \,|\, X > s) = \mathsf{P}(X > t),$$

for $s, t \geq 0$. Can we see this property in the data obtained by simulation?

**1.21.** There is a nice `numpy` method to select (or filter) data that satisfies a certain property. Here, to select all inter-arrival times in the list `a` that are larger than some $s$, we can use the code

```
a = a[a>s]
```

With this, modify one of the above functions in which we compared the emperical distribution of the data to the theoretical distribution. Take $s = 0.5$.

**1.22.** As is apparent from the previous exercise, for small $s$, such as $s = 0.5$, the data satisfies the memory. But what happens when the threshold $s$ becomes larger, e.g., 2?

**1.23.** When $s = 2$ the memoryless property does not seem to hold. Suppose we take more data points, in other words, let's take $N = 10^5$, and see what happens.

**1.24.** What can you conclude from this analysis?

## 1.7 *The number arrivals in an interval and the Poisson distribution*

Besides the memory-less property, we can also analyze whether our inter-arrival times satisfy the 'Poisson property'. That is, when the inter-arrival times are exponential with arrival rate $\lambda$, the number of observations in an interval of a fixed length $t$ should be Poisson distributed with mean $\lambda t$. In this final section we will try to see whether this 'Poisson' property holds for our patient inter-arrival times.

**1.25.** With the code below we can compute the number of arrivals per interval for the list `A` that contains the arrival times of patients. Then we compare the emperical distribution of the number of arrivals in an interval and compare it to the Poisson distribution with the same mean. Explain the code, and then run it.

```
def poisson_1():
    N, L = 10, 1000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    # G = norm(loc=5, scale=1)
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2
```

[3] When testing code it is also a good idea to see what happens if you use bogus numbers. The program should fail or give very strange results.

```
    mean = 20  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
    # plt.pause(10)
    plt.show()

poisson_1()
```

**1.26.** Run the code for various choices of `mean`, for instance, take `mean=5`, and then compare this to a case with `mean=20`. You should notice that when `mean` is large, 20 or so, the variance of the number of arrivals is quite a bit smaller than the variance of the Poisson distribution. Thus, eventually, it becomes clear that the inter-arrival times are more regular than exponential. All in all, we see that data analysis is not simple.

**1.27.** As a final check on our implementation, we can generate exponentially distributed inter-arrival times rather than the uniformly distributed ones we have been using all the time. Implement this idea, and check the results.

## 1.8 *Summary*

**1.28.** Make a summary of what you have learned from this tutorial.

HINTS

**h.1.15.** Note that the doctor sees the combined arrival process, so first find a way to merge the arrival times of the patients into one arrival process as observed by the doctor. Then convert this into inter-arrival times at the doctor.

SOLUTIONS

**s.1.1.** 1. Generate realizations of a uniformly distributed random variable representing the inter-arrival times of one patient.

2. Plot the inter-arrival times.
3. Compute the (empirical) distribution function of the simulated inter-arrival times.
4. Plot the (empirical) distribution function.
5. Generate realizations of uniformly distributed random variables representing the inter-arrival times of multiple patients, e.g., 3.
6. Compute the arrival times for each patient.
7. Merge the arrival times for all patients. This is the arrival process as seen by the doctor.
8. Compute the inter-arrival times as seen by the doctor.
9. Plot these inter-arrival times.
10. Compare to the exponential distribution function with a suitable arrival rate $\lambda$.

**s.1.2.** We put the algorithm in a function so that we can use it later. The algorithm is useful to study, but it has some weak points. In the exercises below we will repair the problems.

We run a test at the end.

```
def cdf_simple(a):
    a = sorted(a)

    # We need the support of the distribution. For this, we need
    # to start slightly to the left of the smallest value of a,
    # and stop somewhat to the right of the largest value of a. This is
    # realized by defining m and M like so:
    m, M = int(min(a)), int(max(a)) + 1
    # Since we know that a is sorted, this next line
    # would be better, but less clear perhaps:
    # m, M = int(a[0]), int(a[-1])+1

    F = dict()  # store the function i -> F[i]
    F[m - 1] = 0  # since F[x] = 0 for all x < m
    i = 0
    for x in range(m, M):
        F[x] = F[x - 1]
        while i < len(a) and a[i] <= x:
            F[x] += 1
            i += 1

    # normalize
    for x in F.keys():
        F[x] /= len(a)

    return F

def test1():
    a = [3.2, 4, 4, 1.3, 8.5, 9]
```

```
    F = cdf_simple(a)
    print(F)
    I = range(0, len(a))
    s = sorted(a)
    plt.plot(I, s)
    plt.show()

"""
You can run a separate test, such as test1 by uncommenting the line
below, i.e., remove the # at the start of the line and the space, so
that the line starts with the word "test_1()". Once the test runs, you
can comment it again, and move to the next test. Uncomment that line,
run the program, comment it again, etc.
"""

test1()
```

**s.1.3.** We have to guess the support of $F$ (the set of points where $F$ makes the jumps) upfront, and we concentrated the support of $F$ on the integers. However, in general $F$ can make jumps at any real number, for instance at 3.2.

```
    # Include this void code to keep the numbering the same between the tutorial text and the jup
```

**s.1.4.** In the answer we let the computer do all the work.

```
def test1a():
    a = [3.2, 4, 4, 1.3, 8.5, 9]
    I = range(0, len(a))
    s = sorted(a)
    plt.plot(I, s)
    plt.show()

test1a()
```

**s.1.5.** Here is one way.

```
def cdf_better(a):
    n = len(a)
    y = range(1, n + 1)
    y = [z / n for z in y]  # normalize
    x = sorted(a)
    return x, y


def test2():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf_better(a)

    plt.plot(x, y)
    plt.show()

test2()
```

**s.1.6.** The reason is that at $s_1$ the first observation occurs. Hence, the unnormalized $F$ should make a jump of at least one at $s_1$. Next, the `range` function works up to, but not including, its second

argument. Hence (in code), `range(10)[-1]/10 = 0.9`, that is, the last element `range(10)[-1]` of the set of numbers $0,1,\ldots,9$ is not 10. Hence, when we extend the range to `len(a)+1` we have a range up to and including the element we want to include.

```
    # Include this void code to keep the numbering the same between the tutorial text and the jup
```

**s.1.7.** The following code is much, much faster, and also very clean. Note that we normalize `y` right away.

```
def cdf(a):  # the implementation we will use mostly. It is simple and fast.
    y = np.arange(1, len(a) + 1) / len(a)
    x = np.sort(a)
    return x, y
```

**s.1.8.** With the `drawstyle` option:

```
def make_nice_plots_1():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf(a)

    plt.plot(x, y, drawstyle="steps-post")
    plt.show()

make_nice_plots_1()
```

**s.1.9.** This is better.

```
def make_nice_plots_2():
    a = [3.2, 4, 4, 1.3, 8.5, 9]

    x, y = cdf(a)

    left = np.concatenate(([x[0] - 1], x))
    right = np.concatenate((x, [x[-1] + 1]))

    plt.hlines(y, left, right)
    plt.show()


make_nice_plots_2()
```

There we are!.

**s.1.10.** Here it is.

```
def cdf_fastest(X):
    # remove multiple occurences of the same value
    unique, count = np.unique(np.sort(X), return_counts=True)
    x = unique
    y = count.cumsum() / count.sum()
    return x, y
```

**s.1.11.** Copy this code and run it.

```
from scipy.stats import uniform

# fix the seed
```

```
scipy.random.seed(3)

def simulation_1():
    L = 3  # number of interarrival times
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = G.rvs(L)
    print(a)

simulation_1()
```

**s.1.12.** Add this code to the other code and run it.

```
def simulation_2():
    N = 1  # number of customers
    L = 300

    G = uniform(loc=4, scale=2)
    a = G.rvs(L)

    plt.hist(a, bins=int(L / 20), label="a", density=True, cumulative=True)
    plt.title("N = {}, L = {}".format(N, L))

    x, y = cdf(a)
    plt.plot(x, y, label="d")
    plt.legend()
    plt.show()

simulation_2()
```

**s.1.13.** Add this to the other code and run it.

```
def KS(X, F):
    # Compute the Kolmogorov-Smirnov statistic where
    # X are the data points
    # F is the theoretical distribution
    support, y = cdf(X)
    y_theo = np.array([F.cdf(x) for x in support])
    return np.max(np.abs(y - y_theo))

def simulation_3():
    N = 1  # number of customers
    L = 300

    G = uniform(loc=4, scale=2)
    a = G.rvs(L)

    print(KS(a, G))

simulation_3()
```

**s.1.14.** Add this to the other code and run it.

```
from scipy.stats import expon

def plot_distributions(x, y, N, L, dist, dist_name):
    # plot the empirical cdf and the theoretical cdf in one figure
```

```
    plt.title("X ~ {} with N = {}, L = {}".format(dist_name, N, L))
    plt.plot(x, y, label="empirical")
    plt.plot(x, dist.cdf(x), label="exponential")
    plt.legend()
    plt.show()

def simulation_4():
    N = 1  # number of customers
    L = 300

    labda = 1.0 / 5  # lambda is a function in python. Hence we write labda
    E = expon(scale=1.0 / labda)
    print(E.mean()) # to check that we chose the right scale
    a = E.rvs(L)

    print(KS(a, E))
    x, y = cdf(a)
    dist_name = "U[4,6]"

    plot_distributions(x, y, N, L, E, dist_name)

simulation_4()
```

It is pretty obvious why these graphs must be different: we compare a uniform and an exponential distribution.

**s.1.15.** The following steps in code explain the logic.

```
def compute_arrivaltimes(a):
    A=[0]
    i = 1
    for x in a:
        A.append(A[i-1] + x)
        i += 1
    return A

def shop_1():
    a = [4, 3, 1.2, 5]
    b = [2, 0.5, 9]
    A = compute_arrivaltimes(a)
    B = compute_arrivaltimes(b)

    times = [0] + sorted(A[1:] + B[1:])
    print(times)

shop_1()
```

**s.1.17.** Add this to the other code and run it.

```
def shop_3():
    N, L = 3, 100
    G = uniform(loc=4, scale=2)
    a = superposition(G.rvs((N, L)))

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))
```

```
    print(E.mean())

    x, y = cdf(a)
    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))  # Compute KS statistic using the function defined earlier

shop_3()
```

**s.1.18.** Add this to the other code and run it.

```
def shop_4():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))
    print(E.mean())

    x, y = cdf(a)
    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

shop_4()
```

This is great. Already for $N = 10$ patients we see that the exponential distribution is a real good fit for the inter-arrival times as observed by the doctor.

**s.1.19.** Add this to the other code and run it.

```
from scipy.stats import norm

def shop_5():
    N, L = 10, 100
    G = norm(loc=5, scale=1)
    a = superposition(G.rvs((N, L)))

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    x, y = cdf(a)
    dist_name = "N(5,1)"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

shop_5()
```

Clearly, whether the distribution of inter-arrival times of an individual patient are uniform, or normal, doesn't really matter. In both cases the exponential distribution is a good model for what the doctor sees. We did not analyze what happens if we would merge patients with normal distribution

and uniform distribution, but we suspect that in all these cases the merged process converges to a set of i.i.d. exponentially distributed random variables. [4]

**s.1.20.** Since $\sigma = \mu = 5$, about 15% of the 'inter-arrival' times should be negative. This is clearly impossible.

```
# keep the numbering
```

**s.1.21.** Here is the answer.

```
def memoryless_1():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 0.5 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_1()
```

**s.1.22.** Here is the answer.

```
def memoryless_2():
    N, L = 10, 100
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 2 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_2()
```

**s.1.23.** Here is the answer.

[4] For the mathematically inclinded, can we prove such a property.

```
def memoryless_3():
    N, L = 10, 100_000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    a = superposition(G.rvs((N, L)))

    s = 2 # threshold
    a = a[a>s] # select the interarrival times longer than x
    a -= s  # shift, check what happens if you don't include this line.
    x, y = cdf(a)

    labda = 1.0 / 5
    E = expon(scale=1.0 / (N * labda))

    dist_name = "U[4,6]"
    plot_distributions(x, y, N, L, E, dist_name)

    print(KS(a, E))

memoryless_3()
```

**s.1.24.** Suppose we would have set $s = 6$. Since the inter-arrival times of an individual patient are $\sim U[4,6]$, there cannot be any inter-arrival that is larger than 6. Thus, we cannot expect that the tail of the expontial distribution (which has an infinite support) will be a good model for the tail of our emperical distribution. In general, (as far as I know) using data to estimate tail probabilities is pretty sensitive, and one should be very careful to use data for this.

Also, it is important to know how data is obtained (and stored) before one starts with data analysis.

```
# keep the numbering
```

**s.1.25.** In step 1 we compute the arrival times of individual patients. In step 2 we merge these arrival times into a sorted set of arrival times as seen by the doctor. For step 3 I specify the mean first. Then, I want that each bin contains about this mean number of arrivals. Since there are $NL$ arrivals in total, I want the number of bins to be equal to `NL/mean`.

The documentation of `np.bincount` and `np.histogram` explain how to use these functions.

**s.1.26.** For `mean=5` we do get acceptable results.

```
def poisson_2():
    N, L = 10, 1000
    G = uniform(loc=4, scale=2)  # G is called a frozen distribution.
    # G = norm(loc=5, scale=1)
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2

    mean = 5  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
```

```
    # plt.pause(10)
    plt.show()

poisson_2()
```

**s.1.27.** Run this code to see the resemblance we have been looking for. It seems that our code satisfies this important consistency check.

```
def poisson_3():
    N, L = 10, 1000
    labda = 5
    G = expon(scale=1.0 / (N * labda))
    A = np.cumsum(G.rvs((N, L)), axis=1) # step 1
    A = np.sort(A.flatten()) # step 2

    mean = 20  # specify the number of arrivals per interval
    bins = N * L // mean # step 3
    p, x = np.histogram(A, bins)

    P = np.bincount(p)
    P = P / float(P.sum())  # normalize
    support = range(p.min(), p.max())

    plt.plot(P)
    plt.plot(support, scipy.stats.poisson.pmf(support, mean))
    # plt.pause(10)
    plt.show()

poisson_3()
```

**s.1.28.** Here are some ideas you should have learned.

1. Algorithmic thinking, i.e., how to chop up a computational challenge into small steps.
2. An efficient method to compute the empirical distribution function
3. The Kolmogorov-Smirnov statistic
4. The empirical distribution of a merged inter-arrival arrival process converges, typically, super fast to an exponential distribution. Thus, inter-arrival times at doctors, hospitals and so on, are often very well described by an exponential distribution with suitable mean.
5. The convergence is not so sensitive to the distribution of the inter-arrival times of a single patients. For the doctor, only the population matters.
6. Using functions (e.g., `def compute(a)`) to document code (by the function name), hide complexity, and enables to reuse code so that it can be applied multiple times. Moreover, defining functions is in line with the extremely important Don't-Repeat-Yourself (DRY) principle.
7. Coding skills: python, `numpy` and `scipy`.
8. Data analysis is tricky, and you need different approaches to obtain information about the data.